Lecture 7

- Static, global variables
- Arrays
  - Declaration
  - Initialization
  - Multi-dimensional arrays
- Function Parameters

Static Local Variables

```
int do_something() {
   static int numberOfCalls = 0; // assignment only
   ...                           // done once!!!
   ++numberOfCalls;

   if ((numberOfCalls % 100) == 0) {
      // print statistics every 100th call
      ...
   }
}
```

- **`static`** modifier
- These variables are initialized before the function is called for the first time
- They keep their values between calls!

Global Variables

- Declared **outside of any block**
- Numbers **initialized** with default value 0
- Scope is entire program unless the **`static`** modifier is used to indicate that the variable's scope is local to the current file
- **Should be avoided** because of potential name conflicts and accidents (every program part can change global variables!)

Global Variable Examples

```
int global;                    // initialized with 0 (*)

float global_pi=3.1415926535; // everyone can change it!

const float global_e=2.718;   // const prevents this!

static int I_am_local_to_the_current_file;


int main()
{
  float global;     // (**), masks (*), uninitialized

  global = 5;       // changes local variable (**)
  global_pi = 0.0;  // possibly not intended
}
```

Array Overview

- Arrays group together variables or constants of identical type. E.g. 8 integers: `int a[8];`
- Access by index `a[i] = 0;`

Array elements are stored in consecutive memory locations (`sizeof(int)= 4`)

```
address   contents

x   ..x+3  a[0]
x+4 ..x+7  a[1]
x+8 ..x+11 a[2]
x+12..x+15 a[3]
x+16..x+19 a[4]
x+20..x+23 a[5]
x+24..x+27 a[6]
x+28..x+31 a[7]
```

This array occupies 8*4=32 bytes in memory

Example

```
// compute average of N numbers

const int N = 10;
double a[N];          // N numbers stored here

int main()
{
   cout << "enter " << N << " numbers" << endl;
   // read N numbers
   for (int i=0; i < N; ++i) cin >> a[i];

   // add them up
   double s = 0;
   for (int i=0; i < N; ++i) s += a[i];

   s /= N;
   cout << "the average is " << s << endl;

   return 0;
}
```


Array Declaration

```
int N;
const int M = 256;

char A[12];    // OK - 12 characters A[0]..A[11]

int B[N];      // not OK! not a constant expresssion

float C[2*M];  // OK - 512 floats C[0]..C[511]
```

- Syntax:
  <type> <indent>`[`<constant-int-expression>`];`
- Integer expression defines the number of objects in the array. They are not initialized!
- Array index always starts with 0

Array Initialization

```
int  A[4];          // 4 integers - not initialized!
int  B[4] = { 4, 3, 2, 1 };   // B[0]=4,..B[3]=1
char C[2] = { 'a','b','c' };  // invalid! too many
char D[]  = { 'a','b','c' };  // OK, declares D[3]
int  E[2] = { 1 };            // OK, E[0]=1 E[1]=0
```

- Syntax:
  <type> <indent>`[`{<const-int-expr>}`]` =
  `{` <const-expr>`,`…`,`<const-expr> `};`
- The list of constant expressions is evaluated and assigned to the array elements
- If list is shorter than array size, 0s are padded.
- Array size can be omitted; it is then defined by the list length

Multi-Dimensional Arrays

- Arrays with more than one index

```
char page[ROWS][COLS];
int table4[2][2][2][2];
```

- Rectangular array of array of ...
- Flat memory layout

```
address   contents

x      :  page[0][0] page[0][1] ... page[0][COLS-1]
x+COLS :  page[1][0] page[1][1] ... page[1][COLS-1]
                        ...
x+COLS*:  page[ROWS-1][0]... page[ROWS-1][COLS-1]
(ROWS-1)
       total: ROWS*COLS bytes
```


Example

```
int table[2][2] = { { 0,1 } , { 2,3 } };
// after initialization:
// tab[0][0] = 0, tab[0][1] = 1
// tab[1][0] = 2, tab[1][1] = 3

int add_table_entries() {

  int s = 0;

  for (int i=0; i < 2; ++i) {

    for (int j=0; j < 2; ++j) {

      s += table[i][j];  // table[i,j] is illegal
    }
  }
  return s;
}
```


Array Access

- Syntax: <ident> [<integer-expression>]
- The expression is evaluated and the array element with that index is accessed
- C/C++ does not check whether the array index is out-of-bounds!

```
#include <cassert>
const int N = 10;
int A[N];

for (int i=1; i <= N; ++i) cout << A[i];
// oops! that's a bug which is hard to detect!

for (int i=1; i <= N; ++i) {  // buggy!
   assert(i >= 0 && i < N); // this kills out-of-
   cout << A[i] << " ";     // bounds bugs dead!
}
```


Arrays as Function Parameters

```
const int N = 10;
int A[N];

void sort(int a[]); // doesn't work, what's a's size?
void sort(int a[], int size);  // makes more sense

...
sort(A, sizeof(A)/sizeof(A[0]));  // OK
```

- Arrays are **passed by reference**
- An array parameter is essentially the array starting address. There is **no size information** attached to it! Need to pass number of elements
- Functions cannot return arrays

Call-By-Value

```
void increment(int x) { ++x; }


int y=5;
increment(y);
// oops, that didn't work: y is still 5!
```

- When a function `A f(B x);` is called:
  ... `f(e)` ...
  expression e's value is copied into the local variable x
- Statements in the body of `f` act on this local copy and do not change values in the evaluated expression e

Call-By-Reference

```
void increment(int &x) { ++x; }

int y=5;
increment(y);     // yup, y now 6
```

- A **reference** to a variable is passed to a function (in form of a memory address)
- Statements in the function body that act on the parameter **change the variable** that has been passed to the function
- Syntax: <call-by-ref-par> ::= <type> &<identifier>
- Side effect + func. can return more than one value
- can only pass variables

Swap Function

```
void naive_swap(int &x, int &y)
{
  x = y;
  y = x;
}

void swap(int &x, int &y)
{
  int temp = x; x = y; y = temp;
}


int a=1, b=2;
naive_swap(a, b);     // oops: a=b=2?!

a=1; b=2;
swap(a, b);           // ok! a=2, b=1
```


Pros & Cons

- **Call-By-Value**:
  + Callee detached from caller, no direct side-effects
  - Data is copied to a local variable. Can be time consuming.
- **Call-By-Reference**:
  - **Side effects**; need to look at function definition to find out!
  - Only variables as parameters
  + No data is copied. Fast access! (const qualifier protects space consuming read-only parameters)